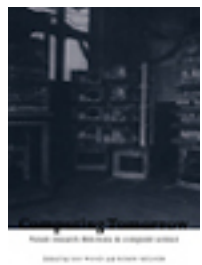
Computing Tomorrow



Computing Tomorrow

Future Research Directions in Computer Science

Edited by Ian Wand, Robin Milner

Book DOI: http://dx.doi.org/10.1017/CBO9780511605611

Online ISBN: 9780511605611

Hardback ISBN: 9780521460859

Paperback ISBN: 9780521103091

Chapter

13 - Semantic Ideas in Computing pp. 246-283

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511605611.014

Cambridge University Press

13

Semantic Ideas in Computing

Robin Milner

13.1 Introduction

Are there distinct principles and concepts which underlie computing, so that we are justified in calling it an independent science? Or is computing a resource or commodity – like water – which is perfectly well understood in terms of existing science, for which we merely have to find more and better uses?

In this essay I argue that a rich conceptual development is in progress, to which we cannot predict limits, and whose outcome will be a distinct science. This development has all the excitement and unpredictability of any science. We cannot predict how the conceptual landscape will lie in a decade's time; the subject is still young and has many surprises in store, and there is no sure way to extrapolate from the concepts which we now understand to those which will emerge. I therefore support my argument by explaining in outline some semantic ideas which have emerged in the last two or three decades, and some which are just now emerging.

I try to present the ideas here in a way which is accessible to someone with an understanding of programming and a little mathematical background. This volume aims to give a balanced picture of computer science; to achieve this, those parts which are mathematical must be presented as such. The essence of foundational work is to give *precise* meaning to formulations of processes and information; clearly, we should employ mathematics in this work whenever it strengthens our analytical power. Thus, rather than avoiding equations, I try to surround them with helpful narrative.

It is a somewhat arbitrary matter to decide when a scientific discipline is mature and stands significantly on its own. Important criteria are that

its ideas have distinctive character compared with those of other disciplines, that these ideas have a well-knit structure, and that they have some breadth of application. As far as distinctive character is concerned, many of the notions discussed here are new; moreover a unifying theme pervades them which we may call *information flow* – not only the volume or quantity of the flow, but the structure of the items which flow and the structure and control of the flow itself. As for breadth of application, there are clear signs that the field of computing spans a wider range than many would have predicted thirty years ago. At that time, the main concern was to *prescribe* the behaviour of single computers or their single programs. Now that computers form just parts of larger systems, there is increasing concern to *describe* the flow of information and the interaction among the components of those larger systems. Another paper in this volume (Gurd & Jones) draws attention to this trend at the level of system engineering and human organizations, while Newman argues that interactive computing should be installed in the mainstream of computing as an engineering discipline. At the fundamental level, the concepts of computing can increasingly be seen as abstractions from the phenomena of information flow and interaction. One may even claim that computing is becoming a science to the extent that it seeks to study information flow in full generality.

In section 13.2, I distinguish between two sources of ideas for computing. One source is the theoretical core of mathematical logic, whose long development (most intense in the last hundred years) makes it a firm basis for our subject. The other source is computing practice itself, as made possible by the digital computer; I claim that this practice is not merely the application of already established science, but also an activity from which scientific concepts are distilled. The remainder of the essay indicates how this occurs, with specific examples; I try to show that software engineering not only exposes the need for a scientific basis, but even provides the germs of the ideas of that science.

In section 13.3, I consider the practical business of making sure that a computing system meets its specification. I pay particular attention to the problem of software maintenance, since the difficulties are most acute there. We see that this activity, and more generally the software design and development process, demand the use of formal descriptions and specifications, and formal methods of analysis and design.

While this need has gained a degree of acceptance in software engineering, I argue in section 13.4 that formalism is not the most important part of the story. Consider the specification of a complete computer sys-

tem S; it is formal because we need rigour, but every formal expression has a meaning, and this meaning has conceptual content. This content will vary from the mundane to the esoteric. The specification of the *complete* system S will express its intended behaviour in terms of concepts drawn from its field of application (e.g. money if the field is banking; geometry and mechanics if the field is robotics); on the other hand the specification of a *component* deeply situated within S will employ concepts which are inherently computational, and pertain to information flow in general.

Section 13.5 is devoted to identifying some of the concepts which have emerged from the challenge to understand conventional programming languages, which provide the raw material for software engineering. Many of these ideas are now stable, and form a considerable part of the foundation of computing. Despite their fundamental character, they have emerged through application experience which has only been possible since the programmed computer was invented. Thus computing is a science which not only informs an engineering discipline, but uses that very discipline as its laboratory for experiment.

Section 13.6 deals with some concepts which are currently emerging, and less stable. Due to the shift of emphasis which I mentioned above from prescribing computer behaviour to describing system behaviour, these emerging concepts are more and more concerned with the interaction of system components, with the flow of information among them, and with the dynamic reconfiguration of the systems to which they belong. Despite the more tentative status of these ideas, they appear to belong to the same large conceptual development as the more stable ideas in section 13.5, and I try to present this continuity of progress.

13.2 The Nature of Computation Theory

At the outset, we should recall that computation has for long had a hard theoretical core closely linked with mathematical logic. Around 1900, the famous mathematician David Hilbert mounted a programme, or issued a challenge, to show that every mathematical truth is deducible – i.e. mechanically derivable – from a set of axioms; the challenge was to find those axioms. This programme was overthrown by the work of Gödel, Kleene, Church and Turing. It was found that any reasonably expressive axiomatic system will inescapably be either *inconsistent*, meaning that one can deduce some proposition and its negation, or *incomplete*, meaning that there are some propositions P for which one

can deduce neither P nor the negation of P. This even occurs in cases where we intuitively regard P as true. In other words, there will always be truths which are not deducible; and this amounts (via the work of these pioneers) to saying that certain mathematical functions cannot be computed by any computer program.

It may appear that this has nothing to do with computing, except to show that it has limitations. But this hard core of computation theory soon ramified into a *classification* of what is computable; that is, there are *degrees* of computability. The ramification began with recursion theory, where it is shown that certain things are essentially harder to compute than others. Within computer science, the theory known as computational complexity carries on this programme; it is reviewed by Atkinson in this volume. We now know that a lot of useful things can be computed in linear time, others take non-linear but polynomial time, still others almost certainly take exponential time and some definitely do. Tasks are also classified in terms of the memory space they require, and in terms of the extent to which parallelism can speed their computation.

Some people, even computer scientists, regard this difficult and unfinished study as essentially the mainstream of computation theory. According to this view, the rest of computer science consists largely of engineering, either of hardware or of software. It is understood that physical science will continue to develop new and more powerful kinds of hardware; on the other hand, software is seen as a resource or commodity whose scientific basis is already established, like water; our task is not to probe further into its nature, but to exploit it by pouring it into everything. It is admitted that this development, or pouring, amounts to a new and intricate engineering discipline; it is hardly allowed that new scientific concepts are needed to underpin the development.

This negative view is at best unproven, as long as we admit that science is concerned with the rigorous understanding of the world about us. There is no doubt that large computer systems exist in the world about us; they are orders of magnitude more complex than anything made before, probably as complex as many natural systems (e.g. ecologies), and their complexity is steadily increasing. Further, there are new kinds of system, containing parts which are not computers and may even be human, which exist only because of the computers they contain; a modern banking system is a good example. In the light of this growth, the negative view seems not only unproven but shortsighted. The surge of interest in object-oriented programming shows that new ideas are entering the very nature of software, though they are imperfectly understood.

Further, as computing is extended to include communication and information flow in distributed and heterogeneous systems, we are faced with the need to understand a newly identified group of phenomena which are indeed pervasive.

In the following sections I argue that this understanding will have broad application; I also show that it employs a well knit body of concepts, some developed in the last few decades in studying programming languages, others newly emerging and more concerned with interaction. Thus the criteria for a significant scientific discipline which I mentioned earlier are met, and the negative view which I have described is refuted. To the extent that the phenomena of computing and communication are man-made, we have a substantial new 'science of the artificial', as Herbert Simon (1980) has recognised.

How do we arrive at these new concepts? In the natural sciences, observation plays a central rôle. A large part of computer science is about man-made phenomena, and we therefore expect engineering practice to play a large part in suggesting new ideas. Computer science is not unique in this; it is hard to imagine that engineering practice (building things) did not play a strong part in the development of 'natural philosophy', i.e. mechanics, including Newton's laws of motion. What is surely unprecedented in computer science is the enormous rate at which we have acquired engineering experience over a few decades. This experience or experiment (in French the word is the same!) is our means of observation.

All experiment is done against an already accepted scientific background, an established body of concepts. In computer science, mathematical logic has always figured largely in this background. (For Turing, the Turing machine and the Ace computer were conceptually close.) This reflects the distinctive part which computer systems play in our environment; they act as extensions of our mental power, in other words as prosthetic devices of the mind. Any prosthetic device is controlled somehow, and in the case of computers the means of control is formal language. Mathematical logic is concerned with the structure and meaning of formal languages, and thus provides essential background to our experiment; it underlies not only *deductive* languages (logics as commonly understood), but also *imperative* or *effective* languages such as programming languages. In studying the way we control systems, or interact with them, we are therefore broadening the field of mathematical logic; we may expect computer science and logic to grow intimately together.

In the next section we shall look at a particular example of systems engineering experience, namely *software maintenance*. The problems encountered there, and our progress in dealing with them, rather clearly illustrate the need for – and the difficulty in arriving at – concepts which help to understand computer systems.

13.3 Understanding a Software System

One of the most important reasons for understanding a software system is to change it. There are two reasons why one may wish to change it; it may be wrong, or it may be required to work in a changed environment. Both of these reasons apply in many cases! But they are not the only reasons for needing to understand software; even if a system is perfect and working in a perfectly unchanging environment, we shall want to build other systems, for different but similar tasks, without starting again – and we can at least hope to use a lot of its pieces unchanged.

What kind of understanding helps in making this sort of adaptive change? This is the problem of *software maintenance*, an activity which is claimed to absorb 80–90% of software effort. Even for experienced computer scientists a simple everyday analogy may be useful in answering this question.

Suppose that you want to adapt your bicycle, when you leave the city to live in the country, so that it will cope with hill paths. How do you know what to change? Your first idea might be to fit fatter tyres which grip the surface better, so you try to find out how to do this. A bicycle is largely open; you can see a lot of the 'code' of a bicycle. Imagine that you've never really looked at any wheeled vehicle properly before; you've only used them. So looking at the bicycle for the first time is like looking at the raw code of a software system. It's all there, but you cannot see all its joins; for example, you can't see whether or how the tyre comes off the wheel – it might be riveted on. In fact you don't even know that there is a separable thing called the 'tyre', a component of a thing called 'wheel'; this structure is not completely manifest.

This problem is met again and again in software engineering, because big systems exist without any, or without enough, structural description. The reason can be completely mundane; perhaps it was written five years ago, by someone who has now left, and written in such a hurry that there was no time to describe it. The only way to solve the problem, if you have not enough time to start again, is what is known as 'reverse engineering'

(Walters & Chikofsky, 1994). At its crudest level this consists in poring over the code for days, weeks or months in order to intuit its structure.

Reverse engineering is important, and it is a serious professional activity. Much has been written on it, and this is not the place to describe it in detail. Because systems engineers are repeatedly confronted with the problem of adapting large inscrutable software systems, it would be pointless to criticize them for using reverse engineering. But we must resist the inference that software engineering will always involve makeshift activity, which – in its crude form – reverse engineering undoubtedly is. More positively, we should try to arrive at means by which this activity will be found unnecessary.

To get an idea of how to avoid reverse engineering, it is useful to take the bicycle analogy two steps further. First, the situation is not usually so bad as suggested above; instead of poring over unstructured low-level code, the reverse engineer will often pore over code written in a high-level language such as Ada or Pascal. This makes the job a lot easier; the modular structure of the system is manifest in the code because these languages are designed to express precisely the interfaces between system components. This is like not only having the bike to look at, but also having the exploded assembly diagram of the bike, showing how the tyre fits on the wheel, and the wheel on the bike.

With this knowledge, perhaps you fit tougher tyres and take the bike on a hill-path. It then collapses; the wheels buckle and the spokes break. Why? The reason cannot be found in the exploded assembly diagram (i.e. the high-level program text). What is missing is any information about the bike *in action*; for example, about the stress which can be transmitted from the road through the tyre (now tough enough) to the wheel. In software terms, the high-level code – though it may be beautifully structured – tells you nothing about how a new user interface (for a tougher environment) can place extra demand on an inner software component which it wasn't made to bear.

A good reverse engineer will, all the same, avoid many of these traps. Mostly this will be because he or she has a good feel for the behaviour of software modules and for their interaction; it may also be because the original Ada code was well adorned with useful comments. Equally, the man in the bike shop was not trained in the strength of materials, or in dynamics, but he too could have told you that new tyres would not be enough for hill riding; you would certainly need new wheels and he might even wonder about the frame.

But we must not be reassured by this sort of horse sense. Software is

unboundedly complex; new situations will always arise in which know-how from previous experience is little help. What is needed is a theory of software dynamics, and a conception of the strength of software material. So this is the second step in the development of our analogy.

What kind of thing is the 'strength' of a piece of software? One way of expressing the strength of a piece of steel is

If you apply a force of no more than 100 newtons to it, and let it go, it will spring back.

For a piece of program we may say analogously

If you give it a quadratic equation with coefficients no bigger than 15000, and let it go, it will compute the roots.

Such a statement – call it S – is usually known as a *specification*. It is precise, and it is not a program; many different programs P may satisfy S. A specification S may be called a contract between the designer of P and the designer of the system environment which will use P. The designer of a complete system has a contract to meet with his or her customer; he must not only ensure that his design meets the contract but he must present an argument that it does so, in a way which others can follow. This second requirement more or less forces him to design in a modular way. Then his argument can be modular too; at each level it will show that if a module can rely upon its parts P_i meeting their subspecifications S_i then the module composed of those parts in turn meets its specification.

The point is that reverse engineering just isn't necessary if good enough specifications are available for each module in a software system, because the specification represents just the knowledge that the reverse engineer is trying to recover. If his customer contract changes, this may mean that the subcontract S for P has to be strengthened – replacing the bound 15000 by 20000 say; this in turn means that P itself may have to be reprogrammed.

I do not wish to imply that this is easy. Specifications can be extremely complex and subtle. But academics are working alongside the software industry to bring about this sea-change in methodology. An early success is VDM (Vienna Development Method) (Jones, 1986), a logic-based methodology which originated in IBM's laboratory in Vienna a quarter of a century ago; another notable example is Oxford University's collaboration with IBM, using the medium of the specification language Z (Spivey, 1992). Every effort must be made to develop

the applied mathematics of specifications, because without it software design – and in particular software maintenance – will remain a black art, full of black holes into which human effort and financial resource are inexorably drawn.

13.4 Underlying Concepts

The previous section amounts to a case for the use of *formal methods* in system engineering. The argument is particularly important for the upgrading of reverse engineering into a more rigorous and less chancy exercise; it will be familiar to many readers. In recent years there has been a fairly widespread acceptance that, at least in principle, specifications should be expressed in a formal language and the reasoning about them conducted with computer assistance where possible.

The problem we now confront, which is central to this essay, is: What do these formalisms express? Most people will reject the extreme formalist view that a formalism expresses nothing! Let us first consider the formal specification of a *complete* computing system; that is, a system which is embedded not in another computing system but in some 'real-world' environment. Then most software engineers (who admit formal specification at all) will agree that this top-level specification should not express anything to do with the internal structure of the system, but should be solely concerned with how the system must behave in its environment. (This may include how much power it consumes, and how long it takes, i.e. its costs.) The specification is therefore written in terms of concepts which pertain to the environment – not to computer science. So far then, concepts which are peculiar to computer science won't show up in the specification of a complete computing system, any more than the concept of a magnetic field shows up in the specification of an electrically driven food-mixer.

But if we admit that a specification of a system expresses behaviour using concepts relevant to the system's environment, then this holds also for incomplete systems – i.e. the modules and submodules within complete ones. By any reasonable count there are more such modules than there are complete systems! So most of the time, a system engineer will be dealing with concepts which refer to the internal working of computing systems. We then naturally ask whether concepts with which we are already familiar – from mathematics, logic, or the external world – will be enough for this purpose. If so, then computer science does not

really deserve to be called a science in its own right – though computer systems engineering will still be a distinct branch of engineering.

I believe that an emphasis on *formal methods*, necessary though it is for systems engineering, may lead to a false conclusion: that familiar mathematical notions, such as the calculus of sets and relations, are enough to specify the behaviour of internal modules of a system, and that we get all the understanding we need – as well as the rigour – by wrapping these up in a suitable logic. Therefore, before discussing some of the new semantic ideas in computer science, I shall give two examples to support the need for them. Other examples can easily be found.

First, consider a computing system at the heart of a communications network which allows links between different pairs of sites to be formed, used and broken. (We need not be concerned whether these are voice links, or links for textual traffic.) The external specification of the system will be something like this: The system accepts requests for links to be formed, forms them, allows interaction across them, takes action if they break accidentally, etc. All this can be specified in terms of external behaviour. Now consider a piece of software within the system whose job is to re-route an existing link (perhaps under certain traffic conditions) while it exists, without in any way affecting the interaction occurring on the link. (Two people having a telephone conversation need not be concerned whether their connection is via one satellite or another, or via none, nor should they be able to detect changes of route in mid-call.) How is this piece of software specified? It must be in terms of some concept of *route*. What is this concept? It is not only a concrete route through hardware; it is also mediated by software, so it has abstract elements too. Whatever it is, it is unlikely to be a concept specific to this one application, nor a concept familiar to users of the system. One can only conclude that it is a concept specific to computer science; in fact, it pertains to the structure of information flow.

As a second example, consider a software-development suite for an industrially used computer language – say Ada. Part of this suite of programs is the *compiler* for the language. The compiler – itself a program – translates the Ada language into basic machine code, or perhaps into some intermediate language close to the machine code, so that the machine can execute it. What is the specification which this compiler is to meet? Among other things (e.g. how errors should be reported) the specification must demand that the translation be correct.

What, precisely, does it mean to say that the translation from one language to another is correct? First, this assertion presupposes a pre-

cise (and therefore formal) definition of the behaviour of a program in each language. Recently, there have been several formal definitions of programming languages; Ada itself was the subject of a monumental attempt, partly successful, at formal definition. Therefore we are reaching the situation that the assertion of compiler-correctness is at least about well defined entities. But this success in formal definition is not the whole story. For, as I said above, every formalism expresses something; in this case, the formalism in which the language is defined should express the *behaviour* of computer programs. To assert that the compiler is correct is to say that, in every case, the behaviour of an Ada program is equivalent to that of its translation. But computer scientists are still not agreed upon the nature of these behaviours; *a fortiori*, it is not settled what constitutes *equivalence* among them! This is partly because Ada tasks run concurrently; the concept of *concurrent process* is unsettled, and there are many different candidates for the notion of equivalence of processes.

Even for languages which have no sophisticated communicational features, indeed no concurrency, the study of equivalence of meaning has generated some of the most striking concepts in computer science, which are even new to mathematics. These concepts are not only concerned with programming languages; they are also about the behaviour of computing and communicating systems in general. After all, this is what our languages are supposed to describe! I shall now go on to discuss some of the concepts which are emerging from the need to understand not only the external specification of computing systems, but also what goes on inside them. I hope that the foregoing discussion has convinced the reader that a conceptual framework is needed for this understanding, and that we cannot assume the concepts to be familiar ones.

13.5 Programs and Functions

I earlier alluded to computers as 'prosthetic devices of the mind'. This may be a fanciful description, but it pinpoints exactly what software is. Just as we control any tool, we control computation and use it for our own purposes, and the *means* of control is just programming. One might hope for a once-and-for-all discipline of control; but this leaves out of account the growing variety of the purposes to which we put computing. Few people would now claim that there can be a universal programming language, or even a small number of them, though the claim has been made in the past.

We may, of course, choose to stay with our current repertoire of half-understood languages, and inhabit Babel for ever. A (rather weak) argument for this is that, after all, we are not doing badly; computer systems do more or less what we intended – and if not, we can adapt to them. But the situation will not stay still; we need only look at the current growth of object-oriented languages to see that new, less understood, languages are emerging – and we have no reason to think that this growth of language will cease. Babel is not stable!

A more insidious argument for continuing to inhabit Babel is that we have no choice, for the following reason: The only alternative is to find a way of understanding languages which does not depend upon language, and this is literally impossible; all descriptions must be expressed in some symbolism – and we are back to the impossibility of a universal language.

This argument ignores the power of mathematical understanding. Certainly mathematics uses formalism, but its essence lies in its *ideas* – expressed using minimal formalism. Thus we seek not a universal *language*, but a universal *understanding* of languages. Indeed, one of the greatest advances in computation theory over the past thirty years has been the development of mathematical theories which explain (i.e. give meaning to) a very wide range of programming languages, including virtually all those which do not involve parallel or interactive computation. With such theories we see the variety among languages in its proper sense – not as prolix symbolism but as variation of methodology within an understood space.

In this essay I cannot do full justice to the theoretical development. But in keeping with my main theme, I want to show how it has led not just to formal or logical definitions, but to a repertoire of concepts which arise *inevitably*, from phenomena of programming which are simple and familiar. So, though we shall end up with a new and significant branch of mathematics, we shall now begin with the most familiar programming ideas, and ask what meaning lies behind them.

13.5.1 Computational domains

We shall look at several well-formed pieces of program, in a familiar kind of programming language. For each one of them, we shall see that it *expresses* or *denotes* an abstract object; for each one we shall ask in what *space* of objects it lies.

Let us begin with a piece of program which declares a procedure for computing the n^{th} power of a real number x:

```
PROCEDURE power(x:REAL, n:INTEGER):REAL
    RESULT = x^n
END PROCEDURE
```

After this declaration, the name `power` has a meaning which it did not have before; it denotes a *function*. But it is not just any function; it is a function which takes objects in one space, the *argument* space, and gives an object in another space, the *result* space. We call the argument space REALS × INTEGERS; it contains all pairs consisting of a real number and an integer. This space is constructed as the *product* (×) of two simpler spaces; the real numbers REALS, and the integers INTEGERS. The result space is just REALS again.

Another way of saying this is that `power` itself denotes an object in a space: the *function space*

$$\text{REALS} \times \text{INTEGERS} \;\rightarrow\; \text{REALS}\ ,$$

where '→' is another way of constructing new spaces from old. In general, for *any* spaces D and E, $D \times E$ is the space containing pairs (one member from D, one from E), while $D \rightarrow E$ is the space of functions from D to E.

It is important to speak generally, in this way, because then we see opportunities for generality of usage. For example, we see that the result space E of the function space $D \rightarrow E$ may itself be a function space; this reflects the fact that in some languages a procedure can deliver a result which is another procedure.

To introduce a third space-construction, we consider how to model a realistic situation, namely that an arithmetic operation can cause overflow; this may make `power` *raise an exception*, i.e. exit abnormally, rather than return a normal result. To model this we can use *summation* (+) of spaces; we say that `power` denotes an element of the space

$$\text{REALS} \times \text{INTEGERS} \;\rightarrow\; \text{REALS} + \{\text{overflow}\}\ ,$$

meaning that it will return sometimes a real number, and at other times the special result 'overflow'. (In general, + can be used for richer alteratives.)

As we consider further program features, we find phenomena which determine more precisely the nature of these spaces. Consider first the

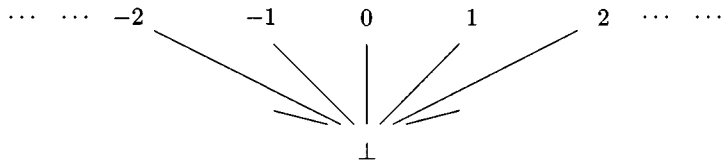


Fig. 13.1. The flat domain of integers

execution of a piece of program which sometimes loops (fails to terminate). We have to give meaning to such programs, even if people shouldn't write them, since we have to describe bad as well as good programs. Suppose someone writes

```
PROCEDURE power0(x:REAL, n:INTEGER):REAL
    IF n>0 THEN ...
    RESULT = x^n
END PROCEDURE
```

where '...' is some looping program; then `power0` no longer denotes a *total* function (one which always gives a result), but a *partial* one. Partiality can be represented by allowing domains to contain a spurious element $\perp$ (pronounced 'bottom'); we write $D_\perp$ for the domain $D+\{\perp\}$, and then we can allow for looping by saying that `power0` denotes an element of the function space

$$\text{REALS} \times \text{INTEGERS} \rightarrow \text{REALS}_\perp,$$

meaning that when applied it will sometimes 'produce' the result $\perp$ – i.e. it will loop without giving any result.

This element $\perp$ is special; intuitively, it is 'less well defined' than every other element. So a space is not, as we might have thought, just a *set* of objects of a certain kind; for its members are *ordered* according to how well defined they are. This ordering is written '$\sqsubset$'; in the space $\text{INTEGERS}_\perp$ for example, we have that $\perp \sqsubset n$ for every integer n, but we don't have either $1 \sqsubset 2$ or $2 \sqsubset 1$ because no integer is *better defined* than any other. We draw $\text{INTEGERS}_\perp$ as in Figure 13.1. It is a very 'flat' space (though it does have $\perp$ below everything else), but more interesting spaces quickly arise. Consider $\text{REALS} \rightarrow \text{REALS}_\perp$; this space contains functions, and we can say that one function is less defined than ($\sqsubset$) another if it gives a defined result less often. Thus the ordering of functions is defined in terms of the ordering of $\text{REALS}_\perp$; writing $\sqsubseteq$ to

stand for '$\sqsubset$ or =', we define $f \sqsubseteq g$ to mean that $f(x) \sqsubseteq g(x)$ for every real number x. The meaning of `power0` is below the meaning of `power` in this ordering; furthermore we can replace the test `n>0` by `n>1`, `n>2` etc., getting functions `power1`, `power2`,... which are better and better defined:

$$\texttt{power0} \sqsubset \texttt{power1} \sqsubset \texttt{power2} \sqsubset \cdots \sqsubset \texttt{power} \ .$$

Thus the meaning of a procedure reflects how ill defined is its *result*. But suppose you give the procedure an *argument* which is ill defined; is it forced to give an undefined result? The following two simple (and stupid) procedures indicate that a procedure may or may not be sensitive to an ill defined argument:

```
PROCEDURE nought(x:REAL):REAL
     RESULT = 0
END PROCEDURE

PROCEDURE zero(x:REAL):REAL
     RESULT = x-x
END PROCEDURE
```

In the space $\textsc{Reals} \rightarrow \textsc{Reals}_\perp$ `nought` and `zero` will denote the same function: that which returns 0 whatever real number you give it as an argument. So this isn't the right space, if we want to reflect that they may differ if you apply them to an ill defined argument! They *do* differ, in some languages. If you execute the two procedure calls

`nought(power0(0,1))` *and* `zero(power0(0,1))`

then one will loop but not the other. The argument to both calls is `power0(0,1)`, and this (if executed) will loop. But the first call will return 0 because the procedure `nought` doesn't *need* to execute its argument, while the second loops (i.e. 'returns' $\perp$) because the procedure `zero` *does* need to work out its argument. We need a space of meanings to reflect this difference; so we assign each procedure a meaning in the space $\textsc{Reals}_\perp \rightarrow \textsc{Reals}_\perp$ (instead of $\textsc{Reals} \rightarrow \textsc{Reals}_\perp$), where they indeed denote *different* functions.

We now go beyond the cosy world of procedures which only compute functions, and consider programs which can change the state (value) of a variable. Suppose that a program consists of a sequence of commands which are executed one by one. One such command is a procedure

declaration, such as we have shown above. Another kind of command is an *assignment* such as

```
z := power(y+z, 2)
```

which changes the value of the variable `z` to be the square of the sum of the (current) values of `y` and `z`. Commands are executed in a *state* in which each name (`y`, `z`, `power`, ...) has a value. So a state is just a function from names to values, and again we can use a space; a state is a member of the space

$$\text{STATES} = \text{NAMES} \to \text{VALUES}$$

where in turn the space VALUES is supposed to contain every kind of meaning which a name can have:

$$\text{VALUES} = \text{REALS}_\perp + \text{INTEGERS}_\perp + \cdots .$$

Now, since each command has the effect of changing the state – i.e. creating a new state from the current one – a command takes its meaning in the space

$$\text{COMMANDS} = \text{STATES} \to \text{STATES} ;$$

for example, the meaning of the assignment `z := z+1` is a function which, given any state $s \in$ STATES, creates a state s' identical with s except that $s'(\mathtt{z}) = s(\mathtt{z}) + 1$.

Let us return to procedures. They may take as arguments and return as results all kinds of values, so we may use the larger space VALUES $\to$ VALUES for their meanings (rather than $\text{REALS}_\perp \to \text{REALS}_\perp$). But, since we are now considering state-change, this space is not enough. For our language may allow assignments such as `z := z+1` to appear *inside* procedures; then each call of a procedure not only returns a value but may also change the state. To reflect this, we take the meaning of a procedure to lie in the more complex space

$$\text{PROCEDURES} = (\text{STATES} \times \text{VALUES}) \to (\text{STATES} \times \text{VALUES}) .$$

As a final twist, we must remember that a state must record not only the association between a variable name `z` and its value, but also the meaning of every procedure name. This means that the space of procedure–meanings has to be included in the space of values, so finally we take

$$\text{VALUES} = \text{REALS}_\perp + \text{INTEGERS}_\perp + \cdots + \text{PROCEDURES} .$$

We have now explored enough programming features. We have captured

the character of each feature by associating with it a specific space of meanings. We have naturally expressed each space in terms of others. The striking thing is that, without dealing with very complex features, we arrived at three spaces – STATES, PROCEDURES and VALUES – each of which depends upon the others; there is a cyclic dependency! Do we have any right to suppose that spaces – whatever they are – can be found to satisfy such equations? The answer is not trivial, and the need for the answer led to the concept of a *domain* (Gunter & Scott, 1990).

13.5.2 Semantic concepts

The preceding paragraphs read like the first few pages of a textbook on programming semantics. But this is essentially the way in which, in 1969, Christopher Strachey identified many of these challenges for a semantic theory of programming. In that year his fruitful collaboration with Dana Scott (Scott & Strachey, 1971) inspired Scott's invention of domain theory. We shall now use the word 'domain' instead of space; we proceed to identify a handful of key concepts which arise inevitably from this short exploration.

Solving domain equations From the simple assumption that every well-formed piece of program should stand for an object in some semantic domain, we have arrived at equations which such domains should satisfy. The exercise would be abortive if there were no solution to these simultaneous equations; we would have failed to find suitable spaces of meanings. But the equations connecting the domains STATES, PROCEDURES and VALUES show a cyclic dependency among them; the existence of solutions is therefore not obvious.

We have to face this fact: If the function domain $D \to E$ contains *all* the functions from D to E, then there *can be no solution* to our equations! This can be shown by a direct adaptation of Cantor's argument that the infinity of the real numbers is strictly larger than the infinity of the integers. For this argument also implies that the size of the function domain $D \to E$ is strictly larger than that of D (if D and E have more than one element); using this fact we can trace around our equations to show that the size of VALUES would be greater than itself!

This paradox is resolved below by making sure that the function domain $D \to E$ actually does *not* contain all functions. This is no contrivance; the omitted functions have no computational significance. Once this functional domain construction is settled, a theory emerges

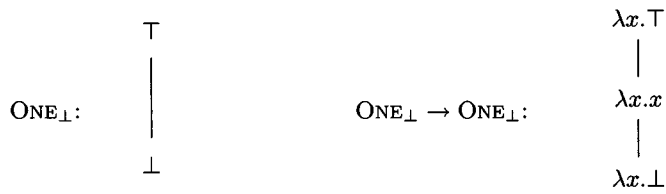


Fig. 13.2. The two-element domain and its function space

which guarantees that *any* family of equations using suitable domain constructions has a solution. Such an equation-set arises in the mathematical description of any programming language; from our simple illustration above it should be clear that it not only lies at the heart of a semantic definition of the language, but gives a clear grasp of the ontology of the language.

Monotonicity From an intuitive understanding of program execution it is easy to accept that, for any procedure which we can define in a reasonable language, any increase in the definedness of its arguments can only lead to an increase (or no change) in the definedness of its result. Such a statement can, indeed, be formally proven if the rules of execution of the language are formally given. We therefore require of the function domain $D \to E$ that it contain not *all* functions, but only those functions f which are *monotonic*, i.e. such that

$$\text{whenever } x \sqsubseteq y \text{ holds in } D, \text{ then } f(x) \sqsubseteq f(y) \text{ holds in } E\ .$$

Take for example the domain ONE with only one element $\top$, so that $\text{ONE}_\perp$ has two elements. Then $\text{ONE}_\perp \to \text{ONE}_\perp$ will not contain the function $f^?$ which exchanges $\perp$ and $\top$, since this function is not monotonic. The omission of $f^?$ from the space is justified because any procedure written to compute $f^?(x)$ from x would have to loop if and only if the computation of x *doesn't* loop, and no procedure could ever be written to do this. The ordering diagrams are shown in Figure 13.2. In the diagram we use λ-*notation* for functions; if M is any expression, then $\lambda x.M$ means 'the function f given by $f(x) = M$'.

Completeness and continuity An interesting domain is one consisting of *streams* (sequences) of some kind of value, say integers. A stream can be finite or infinite. For streams s and s' we define $s \sqsubset s'$ if s' is

a proper extension of s, e.g. $(3,2) \sqsubset (3,2,4)$. The domain is actually given by the equation

$$\text{STREAMS} = (\text{INTEGERS} \times \text{STREAMS})_\perp \ .$$

If $s \sqsubseteq s'$ we say s *approximates* s'. For example, the infinite stream $evens = (0,2,4,\ldots)$ is in the domain, and has an ascending sequence of finite approximants

$$evens_0 \sqsubset evens_1 \sqsubset evens_2 \sqsubset \cdots$$

where $evens_k = (0,2,4,\ldots,2k)$. We naturally think of the 'complete' stream *evens* as the *limit* of this sequence of streams; this notion of limit is mathematically precise, and we write $\bigsqcup x_i$ for the limit of an ascending sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \cdots$ in any domain.

The following fact is (after some thought) intuitive, but needs careful proof: If a function f from streams to integers can be defined by a program, and if $f(evens)$ is well defined – say $f(evens) = 14$ – then there is some approximant $evens_k \sqsubset evens$ for which also $f(evens_k) = 14$. (For $j < k$ we may have $f(evens_j) = \perp$.) In other words: To get a finite amount of information about the result of applying a function to a stream, we only need supply a finite amount of the stream. This is a general phenomenon of information flow in computations; it is not confined to functions over streams. It justifies a further constraint upon the functions f which are included in the domain $D \to E$, that they be *continuous*; this means that for any ascending sequence $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ in D, if $f(x_i) = y_i$ for each i then

$$f(\bigsqcup x_i) = \bigsqcup y_i \ .$$

This condition is usually accompanied by the *completeness* condition that every ascending sequence in a domain has a limit in the domain. These conditions ensure that the function domain $D \to E$ is small enough to avoid the Cantor paradox, and thereby also ensure that solutions exist to the equations among domains which we have met.

Types One of the most helpful concepts in the whole of programming is the notion of *type*, used to classify the kinds of object which are manipulated. A significant proportion of programming mistakes are detected by an implementation which does *typechecking* before it runs any program. Types provide a taxonomy which help people to think and to communicate about programs. Certain type-disciplines have a simple mathematical theory which guides the algorithms used for typechecking;

thus even a simple theory (much simpler than that of the underlying semantics) can lead to good engineering practice.

An elementary example of typechecking is this: by analysing the procedure body, the implementation can discover that the procedure `power` has type `(REAL,INTEGER)->REAL`, as its header line suggests; knowing this, the implementation can disallow the procedure–call `power(2.3,0.5)` because 0.5 isn't an integer.

It is no accident that the type of the procedure 'looks like' a domain. Ignoring the presence of states, the meaning of every procedure lies in the domain $\textsc{Values} \to \textsc{Values}$. The theory of domains provides an elegant story of how the domains $\textsc{Reals}_\perp$ and $\textsc{Reals}_\perp \times \textsc{Integers}_\perp$ 'lie inside' the domain VALUES, and then how the function domain $\textsc{Reals}_\perp \times \textsc{Integers}_\perp \to \textsc{Reals}_\perp$ also lies inside VALUES. Thus we may say that the type-taxonomy of programs is mirrored by a mathematical taxonomy of the 'subdomains' of a domain of values. These remarks only touch upon the rich subject of type-disciplines (Mitchell, 1990), but we have shown that the mathematical framework reflects the computational intuition of types.

Further concepts We cannot treat properly all the computing notions reflected in domain theory, but should mention a few more.

First, some programming languages involve non-determinism, either explicitly by allowing the programmer to choose one of several execution paths at random, or implicitly, e.g. by leaving undefined the order of evaluation of certain phrases (when the order may affect the outcome). There is a domain construction called the *powerdomain* (Plotkin, 1976) of D, written $\mathcal{P}(D)$, whose elements are *sets* of elements drawn from D. If a procedure takes a real argument x and non-deterministically returns either x or $-x$, we can think of it returning the set $\{x, -x\}$; its meaning therefore lies in the domain $\textsc{Reals}_\perp \to \mathcal{P}(\textsc{Reals}_\perp)$.

Second, domains provide an understanding of the flow of *information* in a computation. This is already suggested by our definition of a continuous function; it turns out that any function which can be programmed, since it lies in a function domain, only needs a finite amount of information about its arguments in order to deliver a finite amount of information about its result. Indeed, to say *how much* information is needed in all cases, and *when* it is needed, is tantamount to defining the function! In fact a domain can be seen as a partial ordering of *amounts of information*, and domain theory can be presented entirely in terms

of information flow; domains viewed in this way are called *information systems* (Winskel, 1993).

Third, there is a more subtle point connected with information flow. The elements of a domain which represent finite chunks of information are naturally called *finite*, so what we have seen is that for a finite result you need only compute with finite values. (This is obvious if you are just dealing with numbers, or streams, where the term 'finite' has a familiar meaning. Its meaning is not so familiar when we deal with higher domains.) It also seems to be true, for all computations we can imagine, that even if you are trying to compute an infinite object you can approach this object as close as you like by a succession of finite elements. (Again this is obvious for familiar objects like streams, e.g. the stream of digits in π, but less obvious for higher computational objects like functions.) This property of approximability by finite elements is precisely defined in domain theory, and a domain is said to be *algebraic* if every element has the property. It turns out that all the domains of common interest are algebraic.

Finally, domains have a rich connection with the use of mathematical logic in explaining programming. A well-known method of defining the meaning of a programming language is the so-called *axiomatic method* (Hoare, 1969); a logic is defined in which one can infer sentences of the form $\{P\}A\{Q\}$, where P and Q are logical formulae expressing properties of program variables, while A is a piece of program. The sentence means 'if A is executed with P initially true, then Q will be true when and if the execution is complete'. These *program logics*, as they are called, have special inference rules, and the soundness of these rules can be established by domain theory. A second, more intrinsic, rôle for logic in domains arises from the information flow discussed above; if the quanta of information flowing from argument to result (of a function) are thought of as logical propositions, then the laws of flow can be understood as logical entailments. This leads to yet another presentation of domain theory, known as *domains in logical form* (Abramsky, 1991).

13.5.3 Sequentiality

I shall now discuss an important concept which has been exposed with the help of domain theory, but which may well go beyond that theory.

The reader may not be surprised to learn that, in general, many of the elements in a function domain $D \to E$ are not expressible as the 'meaning' of a procedure in any programming language. It is well-known that

there are just too many (too large an infinity of) functions, if D or E is infinite. But if D and E are both *finite*, we may naturally think that every one of the functions in $D \to E$ can be computed by some procedure. It may seem that the only necessary constraint is monotonicity, which we have already imposed.

But this is not so! To see why, let BOOLS be the set $\{\mathrm{t}, \mathrm{f}\}$, the truth-values, and consider the domain $\mathrm{BOOLS}_\perp = \{\mathrm{t}, \mathrm{f}, \perp\}$. Here is a simple procedure to calculate the 'or' $a \vee b$ of two truth values:

```
PROCEDURE or(a:BOOL, b:BOOL):BOOL
    IF a THEN RESULT = true
         ELSE RESULT = b
END PROCEDURE
```

Because $\mathrm{BOOLS}_\perp$ contains $\perp$, this procedure may express one of several functions in the domain $\mathrm{BOOLS}_\perp \times \mathrm{BOOLS}_\perp \to \mathrm{BOOLS}_\perp$, all giving the usual values when both arguments are defined. Here are three of them, with their differences underlined:

a \ b	t	f	$\perp$
t	t	t	$\underline{\perp}$
f	t	f	$\perp$
$\perp$	$\underline{\perp}$	$\perp$	$\perp$

$a \vee_1 b$

a \ b	t	f	$\perp$
t	t	t	$\underline{\mathrm{t}}$
f	t	f	$\perp$
$\perp$	$\underline{\perp}$	$\perp$	$\perp$

$a \vee_2 b$

a \ b	t	f	$\perp$
t	t	t	$\underline{\mathrm{t}}$
f	t	f	$\perp$
$\perp$	$\underline{\mathrm{t}}$	$\perp$	$\perp$

$a \vee_3 b$

The first function, $\vee_1$, gives $\perp$ if either of its arguments is $\perp$; this is the one expressed by our procedure, if the rule of evaluation requires that both arguments of any procedure-call `or(···,···)` must always be evaluated. But some procedure-call disciplines require an argument to be evaluated only when needed in the execution of the procedure body; in that case our procedure expresses the second function $\vee_2$ because it needs b only when $a = \mathrm{f}$, not when $a = \mathrm{t}$. Note that $\vee_2$ is not symmetric. The third function $\vee_3$ is symmetric (like $\vee_1$), and has been called the 'parallel-or' function.

Thinking in terms of information flow, $\vee_1$ delivers a quantum of information only when *both* arguments deliver one, while $\vee_3$ delivers the quantum t if *either* argument delivers this quantum. How could any procedure `or(a,b)` be written to behave in this way? It must somehow evaluate both its arguments *in parallel* – hence the nickname 'parallel-

or' – since it has no means of knowing which argument may provide a quantum, or of stopping when the first quantum arrives (ignoring or aborting the possibly looping evaluation of the other argument).

In fact no conventional programming language such as Fortran, Algol, Cobol, Lisp or C provides control features with this power. This is not a vague statement; once the evaluation rules of these languages are formally given (as they can be) then one can prove the foregoing assertion. Loosely speaking, the evaluation-discipline of these languages is inherently *sequential*.

When this was first observed, it seemed that using domain theory one should be able to define the notion of *sequential function* mathematically; one could then exclude the non-sequential functions from any function domain $D \to E$, and finally claim that domain theory corresponds perfectly to the evaluation mechanisms of a well defined class of languages – the sequential ones. Large steps have been made in this direction. The so-called *stable* functions (Berry, 1978) are a restricted class of the continuous functions which excludes the 'parallel-or' function, but includes all functions definable by ordinary programs; unfortunately it also includes some which are intuitively non-sequential! On the other hand there is an elegant notion of *sequential algorithm* (Berry & Curien, 1982), more abstract than a program but less abstract than a function. These ideas apparently converge upon the elusive notion of 'sequential function', but they have still not isolated it.

Results have recently been obtained which suggest that, to explain sequentiality properly, one has to deal with ideas not intrinsic to domain theory. An intuition for this can be expressed as follows. Think of the result $z = f(x, y, \ldots)$ of a function being computed in 'demand mode'; that is, if the caller demands more information about z, then f will supply it – if necessary by in turn making demands upon one or more of its arguments $x, y, \ldots$. Several kinds of event occur repeatedly, in some order: The caller may ask f for a quantum of the result z; f may ask any of its arguments $x, y, \ldots$ for a quantum; f may receive a quantum from an argument; f may deliver a quantum to the caller. Then we may say loosely that f is operating *sequentially* if whenever it has requested a quantum from some argument, say x, it will remain idle until that quantum arrives; in particular, it will not request any quantum from another argument.

This is, of course, a very mechanical explanation; the aim of theoretical research in this area is to find concepts of information flow which underlie such mechanical disciplines. Recently, it has been observed that this

sequential discipline is highly significant in *game theory*, and in terms of that theory a new approach to understanding sequentiality (Abramsky *et al.*, 1994; Hyland & Ong, 1994) has been found. While it is too early to judge with certainty, a mathematically convincing account of sequentiality does appear to be emerging.

13.5.4 Summary

My brief survey of semantic ideas for sequential programming is now complete. I spent some time on the idea of sequentiality itself; it is a prime example of an idea which is intrinsic to computing, or to information flow, and which was analysed nowhere in mathematics or logic before computers and their languages forced us to study computation in its own right. The difficulty in isolating sequentiality as a concept has surprised most people who work in semantics; this leads them to expect the emergence of further deep insights into the structure of information flow.

Sequentiality is a pivotal idea, for its understanding will contribute also to the understanding of *parallel* computation; it will help us to describe more clearly what is *lacking* in sequential programming languages, preventing them from expressing a whole class of parallel computations exemplified by the 'parallel-or' function. We now turn to parallel processes.

13.6 Interaction and Processes

With the advent of multiprocessing and computer networks, computer science could no longer comfortably confine itself to the building and understanding of single computers and single programs. This is a formidable activity, but it must be carried on now in the context of the broader field of computing systems – and indeed systems with non-computer (even human) components. Terms such as 'computer', 'program', 'algorithm' and 'function' will be less dominant, beside new terms such as 'system', 'information', 'interaction' and 'process'. All the same, it will be fatal to have one set of concepts for the lower (older) level, and one for the upper (newer) level; we have to expand our ideas and theories, not just increase their number.

In this section I shall try to view these levels, at least for the purpose of discussion, as instances of one kind of structure which I shall call a *communicating system*. We call the behaviour of such a system a

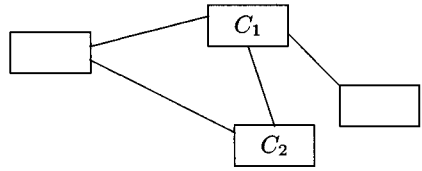


Fig. 13.3. The structural diagram of a system

process, and we shall limit ourselves to *discrete* processes – those whose behaviour is made up of atomic (not continuous) changes of state.

When we build or study a communicating system, we often draw a diagram such as Figure 13.3 to show how it is composed of subsystems such as C_1 and C_2. If we know what happens on the arcs of such a graph, i.e. we know the way in which the components react one with another, then we can build an understanding of the system from an understanding of its components (the nodes). Indeed, such a diagram remains vacuous until we make precise what kind of traffic occurs between components; the nature of this traffic determines the concepts which shape our understanding of a whole system.

We shall now explore a variety of computational models which differ in a fundamental way just because their concepts of traffic between components differ. We shall begin with the *functional* model, which has already been powerfully exploited in understanding programming languages; this topic was explored in section 13.5. In this way we see that the functional model, and the languages based upon it, represents an especially disciplined form of interaction in a hierarchical system. We may then contrast it with models which pay increasing attention to the autonomy of the separate components and their concurrent activity.

13.6.1 The functional model

Let us begin by assuming that a computer program starts by reading in some arguments and finishes by yielding a result, remaining obediently silent in between. This is the situation which we modelled in the previous section. Our model was abstract; we represented a program as a mathematical function, which captures just the external behaviour of a program but says nothing about how it is constructed, nor how it behaves during execution.

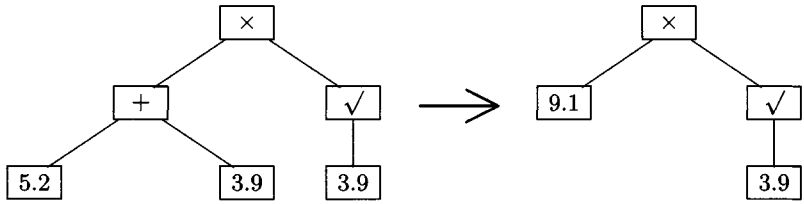


Fig. 13.4. Evaluating an expression

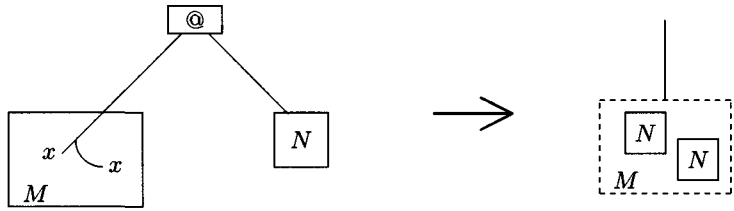


Fig. 13.5. The λ-calculus evaluation rule: $(\lambda x.M[x])@N \longrightarrow M[N]$

But a program does behave in a precise manner, and it serves our purpose to consider it as a communicating system with a certain discipline for traffic between its components. At its very simplest, a program is like an automated desk calculator evaluating complex expressions. We think of the system diagram as a tree, each node as an operator, and the traffic consists of the passage of values – representing completed evaluations – along arcs. A step of evaluation is shown in Figure 13.4.

Though it is hardly apparent from this simple example, highly complex programs can be treated as functions evaluated in this way. An important feature of the functional model – many would call it the essence of the model – allows the very values computed by components to be *themselves* programs, understood as functions. A (still simple) example is the procedure which, for any argument x, will evaluate the expression $(5.2 + x) \times \sqrt{x}$. In λ-notation, which we touched upon earlier, the function computed by this procedure is written $\lambda x.\,(5.2+x) \times \sqrt{x}$. Now, using an operator @ which applies a function to an argument, we can represent procedure-call as an evaluation step as shown in Figure 13.5, where $M[N]$ means that N is substituted for x in M. This is in fact a picture of the single evaluation rule of the λ-*calculus*, the calculus of

functions originated by the logician Alonzo Church (1941). The big box on the left represents the expression $\lambda x.M$ 'wrapped up' as a value; the evaluation unwraps it and makes a tree – so if $M = (5.2 + x) \times \sqrt{x}$ and $N = 3.9$ then the result is the tree for $(5.2 + 3.9) \times \sqrt{3.9}$ and the evaluation continues as in Figure 13.4. In passing, it is worth noting that the evaluation rule for procedures in ALGOL60, known as the *copy rule*, is a special case of λ-calculus evaluation.

Just as we saw that any program or procedure (in a conventional language) can be represented as a value in a domain, so we can represent the *evaluation* of programs and procedures by this hierarchical discipline of communication among the components of a (tree-like) system. The generality here is striking; the control features of conventional languages can *all* be represented as suitable components (nodes) in these trees. The power comes largely from the *higher-order* nature of λ-calculus; the argument x for a function $\lambda x.M$ may itself be a function, and M may contain function applications (@) – precisely as procedures in conventional languages may take procedures as arguments, and their bodies may contain procedure-calls. This challenge to understand programming languages has greatly deepened the study of the λ-calculus in the last two decades (Barendregt, 1984).

The simple hierarchic discipline of interaction which I have described allows very little autonomy for its components; we now proceed to develop it and to increase the autonomy.

13.6.2 The stream model

It may strike the reader that the components, or operators, in our evaluation trees could perform a more interesting rôle, suggested by the information flow which we discussed in section 13.5. We may think of such a tree operating in demand mode; in response to a request for a quantum from its parent node, each node may respond or may transmit requests to one of more of its children. Though still arranged in a hierarchy, the components are now interacting more subtly; indeed, the distinction between sequential and non-sequential computation can be analysed in terms of the tree structure.

If we relax the hierarchy by allowing arbitrary – even cyclic – connectivity among components, we arrive at the *stream model* (Kahn, 1974; Broy, 1987). This is a model of the well-known interaction discipline known as *pipelining*; each arc in a system diagram has an orientation, and is interpreted as a *pipeline*, along which a stream of

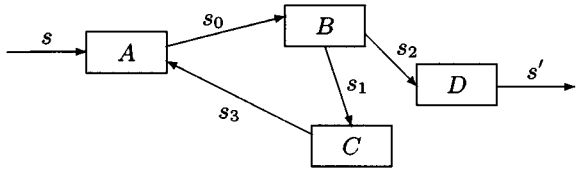


Fig. 13.6. A typical pipeline system

data (quanta) can flow. Moreover, such systems are truly heterarchic, or distributed; no single component maintains central control. A typical system is shown in Figure 13.6. Now let us imagine that a quantum of information is an integer. The total history of the system's behaviour is characterized by the stream of quanta which flow along each of its arcs. But each stream is a finite or infinite sequence of integers, which as we saw in section 13.5 is just an element of the domain

$$\text{STREAMS} = (\text{INTEGERS} \times \text{STREAMS})_\perp \ .$$

A common situation is that each component (node) behaves as a continuous function over STREAMS. In that case, it can be shown that the whole system also behaves as such a function. To find this function, we first write down equations which all the streams satisfy; in this case they are

$$\begin{array}{lllllllll} s_0 & = & A(s, s_3), & s_1 & = & B_1(s_0), & s_2 & = & B_2(s_0), \\ s_3 & = & C(s_1), & s' & = & D(s_2) \ . \end{array}$$

Eliminating s_1, s_2, s_3 yields $s' = D(B_2(s_0))$ where $s_0 = A(s, C(B_1(s_0)))$. The latter equation expresses s_0 in terms of itself and the given s. Domain theory ensures that this 'recursive' equation has a solution for s_0; moreover, there is a solution which is *least*, in terms of the information ordering $\sqsubset$. Moreover, this solution agrees exactly with what is expected from an operational description of stream computations; this agreement provides another justification of the theory.

The stream model not only is mathematically pleasant, but also is realistic for systems in which communication occurs only via 'pipelines'; as this term suggests, this kind of information flow is by no means restricted to computers. The model has stimulated much study, aiming at a more general model. For example, if the behaviour of any node is critically dependent upon which of its input streams supplies quanta more quickly, then the situation is much more delicate. It is also natural to examine the case in which the data structure connecting two nodes

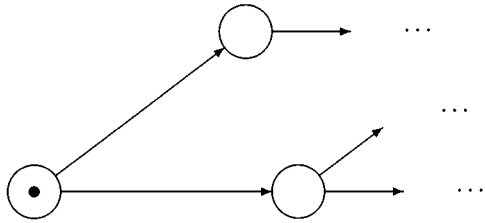


Fig. 13.7. A transition diagram

(i.e. the 'traffic' structure) is other than a sequence. This has given rise to the study of *concrete domains* (Kahn & Plotkin, 1993), which contributed to the growing understanding of sequentiality discussed in section 13.5.

The stream model and its variants are appropriate whenever systems are loosely connected, in the sense that an event at one node need never be synchronized with an event elsewhere.

13.6.3 The reactive model

Let us now contrast the functional and stream disciplines of communication with a more intimate one, where an arc between two components represents their contiguity. Contiguity is a physical concept; a wooden block is contiguous with the table on which it rests, so a movement of the table is immediately felt by the block. But contiguity makes sense with respect to information too. If neighbouring components are a human and a workstation, then the pressing of a key on the keyboard is simultaneously sensed by the human and by the workstation; one can say that they *change state* simultaneously.

One of the earliest ways to model change of state was the basis of *automata theory*, developed mainly in the 50s. The theory made use of *state transition diagrams* such as in Figure 13.7. In these diagrams each circle represents a state, and each arc (arrow) represents a possible transition between states. In the diagram, there are two *alternative* transitions from the left-hand state, and the token • indicates that the left-hand state is the *currently holding* state.

In the early 60s, Carl-Adam Petri (1962) pointed out that automata theory (as it was then) cannot represent the *synchronized* change of state in two contiguous systems. His idea for repairing this weakness is

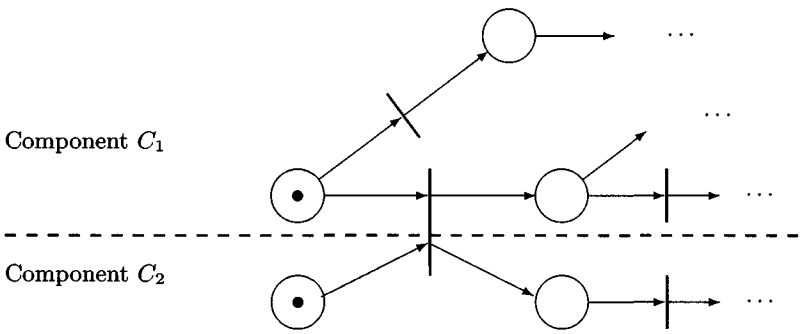


Fig. 13.8. A Petri net, or diagram of shared transitions

simple but far-reaching; it allows transitions which are *shared* between separate components of a system. Suppose that Figure 13.7 represents the possible transitions for component C_1, and we want to indicate that the lower transition of C_1 may not occur independently, but must coincide with a certain transition in component C_2. This would appear as in Figure 13.8. The tokens represent the *concurrent holding* of two states, one in C_1 and one in C_2, and transitions are drawn no longer as arcs but as another kind of node (here a bar); thus a shared transition typically has more than one ingoing (and more than one outgoing) arc, representing *how* it is shared.

This apparently mild development has opened a new and complex world. In this new interpretation the traffic upon the arc between C_1 and C_2 is no longer merely the passage of the result of a (functional) computation by C_1 as an argument to C_2, as it was in the stream model. Behaviour can no longer be expressed just in terms of arguments and results; C_1 in its behaviour requires (or suffers) repeated interaction with its neighbours. The transition shared across the dotted line in the diagram is just one of these interactions.

The behaviour of such communicating systems has been extensively analysed in terms of a handful of important concepts. The most obvious of these is *non-determinism*, or *conflict*. This is present already in Figure 13.7, where either but not both of the two possible transitions takes place. It is unrealistic to expect rules and details (e.g. priorities, timing) to be included in a system model which will determine its behaviour completely.

Allied to non-determinism are other important notions. One such no-

tion is *preemption*, of which Figure 13.8 shows an example; if the upper transition in component C_1 occurs (i.e. the upper token traverses the upper bar) then the lower transition of C_1, though previously possible, is no longer possible. A second important notion is that of *concurrency* or *causal independence*. Two transitions may be called concurrent if they involve no common states; this means that one cannot preempt the other. The two transitions at the right of Figure 13.8 (one in C_1, one in C_2) are concurrent.

The invention of Petri nets and subsequent models represents the slow distillation of concepts from practical experience. An important step in this direction was the concept of *event structure* (Nielsen *et al.*, 1981). In a Petri net each pair of transitions is either *in conflict*, or *concurrent*, or *in causal dependence*; an event structure consists of a relational structure in which these three binary relations satisfy certain axioms. Event structures are somewhat more abstract than Petri nets; the study of stronger or weaker axioms for event structures has contributed to the development of the concept of process.

13.6.4 Process constructions

Much analysis of practical systems has been done using the ideas which we have mentioned. However, when we leave the stream model we are no longer dealing with mathematical functions. We are dealing with operational or behavioural properties of computational processes, but we have no agreed *concept of process* at the same level of abstraction as mathematical functions, nor a *calculus of processes* as canonical as is the λ-calculus for functions. Several steps have been taken in this direction, but the study is far from complete.

An important step was to notice the striking coincidence between a (simple) notion of process and the mathematical idea of a *non-well-founded set* (Aczel, 1988). In mathematical set theory, sets are traditionally well-founded; this excludes, for example, a set S which satisfies a recursive equation such as

$$S = \{a, b, S\} .$$

If S satisfies this equation, then it can be depicted as in Figure 13.9, where the nodes are elements (which may be sets) and an arrow $S_1 \longrightarrow S_2$ means that S_1 contains S_2 as a member – i.e. $S_2 \in S_1$. This 'set' might be written $S = \{a, b, \{a, b, \{a, b, \ldots\}\}\}$; but most of us have been

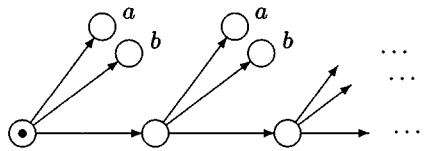


Fig. 13.9. A process which is a non-well-founded set.

so thoroughly 'well-founded' by our education that we think it is cheating to allow it as a solution to the equation, because it has an infinite containment chain $\ldots \in S \in S$. Why are we so suspicious? It is perhaps because we are biassed towards static notions; we think of a set almost as a box *containing* its elements – and of course boxes can't nest infinitely! But think of an arrow representing *procession* rather than containment; think of S as proceeding (non-deterministically) either to the terminal state a, or to the terminal state b, or to a state in which it has exactly those three alternatives again. We may say that a process is constructed from its possible futures, exactly as a set is constructed from its members.

If you object that processes are one thing and sets another, then you may keep them distinct if you wish; but their mathematics has a lot in common. The topic of discrete processes is of growing importance; children in the first decades of the twenty-first century may become as familiar with the mathematics of processes – complete with infinite procession! – as we are with sets. In this way computer science even makes some contribution to the foundations of mathematics.

We shall now look at some other constructions of processes. We may not know what processes *are*, abstractly, but we do know ways of combining them. It is typical in mathematics that the objects of study are nothing by themselves; it is the operations upon them, and the mathematical behaviour of those operations, which give them status. This is the attitude which gave birth to *process algebra* in the late 70s; if we can (with the benefit of practical experience) find some small distinguished class of operations by which interacting systems are built, then to agree upon the algebraic properties of these operations is a necessary step towards the concept of process.

In seeking such operations we may follow the paradigm of the calculus of functions (the λ-calculus), but we must be wary. Consider first the familiar operation of *function application*, @; if M and N are functional objects, then $M@N$ stands for the application of M as a function to the

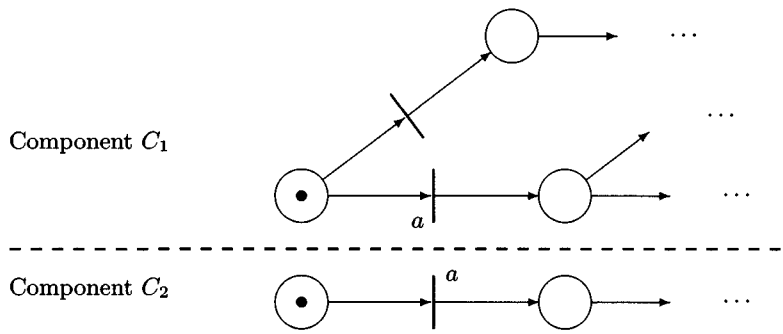


Fig. 13.10. Two components of an interactive system before assembly

argument N. We may look for an analogue of application over processes. But note that application is non-commutative; $M@N$ is quite different from $N@M$.

By contrast, if the arcs of Figure 13.3 are to be interpreted as interaction, which is a *symmetric* concept, then the operation of composing two systems – by connecting them to allow interaction – must be a commutative one. We find a clue to the nature of process composition if we first separate out the two components from which Figure 13.8 was formed, as shown in Figure 13.10. Notice that the separate transitions in C_1 and C_2 are labelled. In Hoare's process algebra CSP and other process algebras (Hoare, 1985; Milner, 1989; Baeten and Weijland, 1990), this labelling dictates where the interactions will occur when we combine C_1 and C_2 by *parallel composition*, $C_1 \parallel C_2$. (This operation is indeed commutative, and also associative.) The result is not quite the system of Figure 13.8, because if we further combine with C_3, forming $C_1 \parallel C_2 \parallel C_3$, then C_3 may also be able to take part in the transition a. If we wish this transition to be shared only between C_1 and C_2, we must write $(C_1 \parallel C_2)\backslash a$. The unary operation $\backslash a$, called *hiding* in CSP, has this localizing effect; it is really just a local declaration of (the name of) a transition. (In passing, this is another example of something distilled from programming practice into a theory: every programmer knows about local variable declarations.)

These two operations, composition and hiding, give us some idea of what process algebra is. Through the 80s there has been considerable progress in using algebra to express and analyse communicating systems, and considerable theoretical development of the algebra. Full agreement

on the algebraic operations has not been reached, but enough to indicate a consensus of understanding.

It is central to the argument of this essay that process algebra is not the application of an *already known* abstract concept to the study of computation. The real situation is more untidy, and far more exciting. The algebraic method acts as a guide along the difficult path towards *discovering* an abstract notion of process which matches the way we construct discrete systems, or which helps us to analyse existing systems. This discovery is occurring incrementally, via definition and experiment, not all at once.

13.6.5 Mobility

When do we have enough operations for constructing processes? That is, when can we claim that we truly know what kind of thing a discrete process is, because we know all the operations by which it can be constructed? This may never occur with certainty; but our algebraic experiments can give evidence that we *lack* enough operations, for example if they do not allow the construction of systems with some desirable property. This has indeed occurred with the property of *mobility*, as we now show.

If we are content to model the behaviour of a concrete interactive system, whose processors are linked in a fixed topology, then we expect the shape of the system graph (Figure 13.3) to remain fixed throughout time. Many communicating systems – software as well as hardware – have a fixed connectivity.

We have already seen one model where the connectivity is flexible – the λ-calculus. It also turns out that the various process algebras explored during the 80s all admit a controlled kind of mobility; they allow that an agent may divide itself into a collection of subagents which are interconnected, or may destroy itself. These capabilities have made it possible to build theoretical models for many real concurrent programming languages, as well as to analyse distributed systems and communication protocols.

But something is lacking in this degree of mobility; it does not allow *new* links to be formed between *existing* agents. This simple capability is, in fact, crucial to the way in which we think about many systems of widely differing kinds. Consider a multiprocessing system supporting a parallel programming language; if we model the implementation, we have to consider the mechanism by which procedures of the (parallel) program

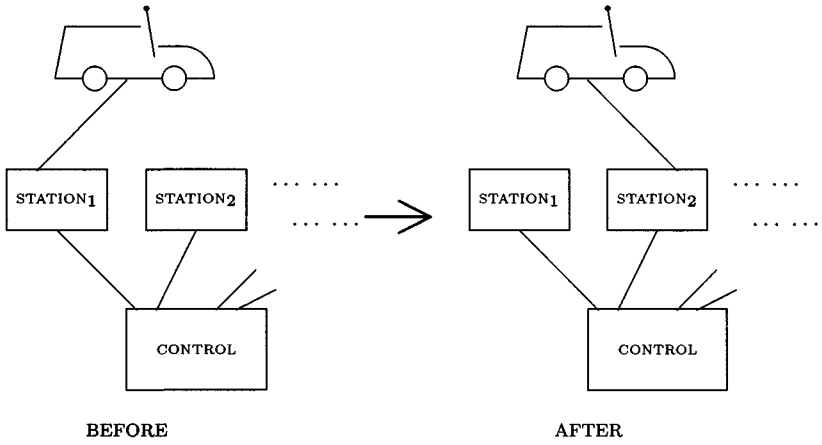


Fig. 13.11. A mobile telephone network

are relocated upon different processors, to achieve load balancing. If we represent both the procedures of the program and the processors of the machine as agents in the model, then relocation of procedures is an instance of mobility among agents. As a second example, consider the resources allocated by an operating system to different jobs; this allocation varies dynamically, and is naturally modelled by the creation and destruction of links. Third, consider the cars of a mobile telephone network; as a car moves about the country it will establish (by change of frequency) connection with different base stations at different times. This is shown in Figure 13.11.

These examples suggest that dynamic reconfiguration is a common feature of communicating systems, both inside and outside the computer. The notion of a *link*, not as a fixed part of the system but as a datum which it can manipulate, is essential to the understanding of such systems. Indeed, we need the notion at the very basic level of programming, when we consider data structures themselves. If a data structure is mutable (and most realistic programs deal with at least some mutable data) then we may consider it to be a process with which the program interacts. This makes particularly good sense if it is a structure which is shared among subprograms running in parallel. An instance of dynamic reconfiguration occurs when one subprogram sends the *address* of a mutable data structure to another subprogram.

We then naturally ask: Is there a common notion of link which subsumes *pointers*, *references*, *channels*, *variables* (in the programming

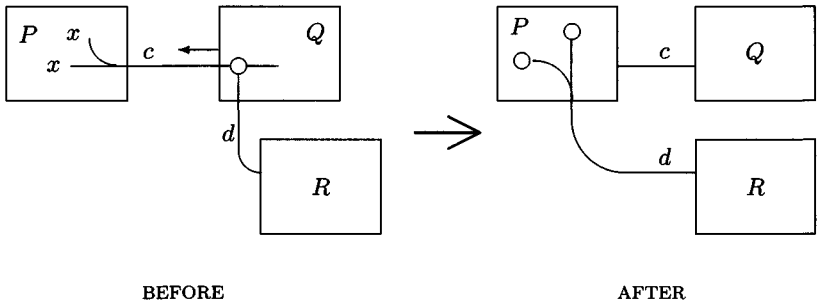


Fig. 13.12. The π-calculus interaction $cx.P[x] \parallel \overline{c}d.Q \longrightarrow P[d] \parallel Q$

sense), *locations*, *names*, *addresses*, *access rights*, ..., and is it possible to take this notion as basic in a computational model? To put it another way: What is the mathematics of linkage?

In searching for this, it is quite reasonable to look first for a *calculus* which treats linkage and mobility as basic, in much the same way as the λ-calculus treats function application as basic. For the λ-calculus was invented and put to considerable use before it was understood in terms of abstract functional models such as domain theory. A recent calculus for mobile processes is the *π-calculus* (Milner *et al.*, 1992); it is in some ways more primitive than the λ-calculus (which can indeed be embedded in it), and takes interaction at a link as its *only* basic action. This is not the place to give its details, but some idea of it is given by the diagram in Figure 13.12. To understand the diagram, let us compare its constructions with those of the λ-calculus and its evaluation mechanism, as shown in Figure 13.5. In the λ-calculus, the *function* $\lambda x.M[x]$ can receive any argument N and then compute $M[N]$; in the π-calculus, the *process* $cx.P[x]$ can receive along the channel c any channel d, and then proceed as the process $P[d]$. The π-calculus also has the process $\overline{c}d.Q$, which sends the channel d along the channel c and then proceeds as Q. Finally, in place of the application @ of a function to an argument, the π-calculus (like CSP) has parallel composition $\parallel$ of two processes. We may now compare *evaluation* in λ-calculus with *interaction* in π-calculus; in Figure 13.5 the agent N is moved into the places held in M by the variable x, while in Figure 13.12 it is not the agent R but a *link* d to R which moves; it travels along a link c and occupies the places held by x in P.

This calculus is simple, and it directly models a number of phenomena. The three examples discussed earlier – including the mobile telephone

network – yield to it, and it has also been used to model the exchange of messages in concurrent object-oriented programming. The calculus thus appears to be sufficiently general; if its technical development is satisfactory (here the λ-calculus provides a valuable paradigm), then it will provide strong evidence that mobility is an essential part of any eventual concept of process.

13.6.6 Summary

We have explored different computational models in terms of the idea of interaction; emphasis is placed upon concurrent processes, since interaction presupposes the co-existence of active partners. In the previous section, we found that the domain model can be understood in terms of amounts of information, and also that *sequential* computation corresponds to a special discipline imposed upon the flow of information. In the present section, we have found that a key to understanding *concurrent* or interactive computation lies in the structure of this information flow, i.e. in the mode of interaction among system components; this attains even greater importance in the case of mobile systems where the linkage does not remain fixed, a case which arises naturally both in computer systems and in the world at large.

There is no firm agreement yet on the properties which an *abstract* concept of concurrent process should possess. I hope to have shown that we may advance towards the concept by distillation from practice, via models which are somewhat *concrete* and *formal*, but nonetheless mathematical. The formalisms of computer science are not only important as a vehicle for rigorous description and verification of systems; they are also a means for arriving at a conceptual frame for the subject.

13.7 Conclusion

At the beginning of this essay I argued that the science of computing is emerging from an interaction between mathematics, especially logic, and practical experience of system engineering. I discussed the difficulties experienced in software maintenance, a branch of system engineering which clearly illustrates the need for formality and conceptual clarity in software design. I stepped from there into the realm of software itself, i.e. the programming languages in which software is written; the remainder of the essay has been devoted to explaining concepts which underlie software, since these are at the core of the emerging science.

Before concluding this discussion of concepts, let us put them into perspective. In July 1994, the British press reported yet another fiasco in computer system engineering; a failed design for a system for the Department of Social Security, which may cost the tax-payer fifty million pounds. Software engineers are likely to blame the failure on mis-application of the *software development process*, the managerial and technical procedures by which the specification of software is elicited and its manufacture conducted. They will be wrong to lay the blame wholly at this door; this would be to presume that software is, like water, a commodity which certainly demands a high standard of engineering and management, but which is well understood in itself. I hope to have shown in the preceding two sections how ill founded this presumption is. I believe that advances in software science will change the very nature of software engineering processes; the obsolescence of reverse engineering (discussed in section 13.3) will be one example.

Throughout the essay I have discussed semantical concepts with some degree of informality. In some cases, particularly in section 13.6, the informality is *necessary* because the concepts themselves have not yet been fully distilled from practice; this is particularly true of the concept of process. In other cases there is a well developed theory (e.g. domain theory), and the informality is *possible* because the concepts are intuitively compelling, in the same way that the notion of force in mechanics is compelling. It is a test of the scientific maturity of a discipline that its conceptual framework is robust enough to allow precise but informal communication of ideas. On this criterion, computer science has made strong progress, but has much further to go.

In the introduction two other criteria of scientific maturity were mentioned: broad application, and a well knit body of ideas. I have shown that these two criteria are being met, not separately, but by two trends which are closely linked. First, the field of application of computer science has enlarged from the study of single computers and single programs to the far broader arena of communicating systems; so increasingly the field is concerned with the flow of information. To match this, we have seen in sections 13.5 and 13.6 above a progression in the way computation theories are presented, leading from the notions of *value*, *evaluation* and *function* towards notions of *link*, *interaction* and *process*. Thus both applications and theories converge upon the phenomena of information flow; in my view this indicates a new scientific identity.